



i n v e n t

IA-64

IA-64 Architecture

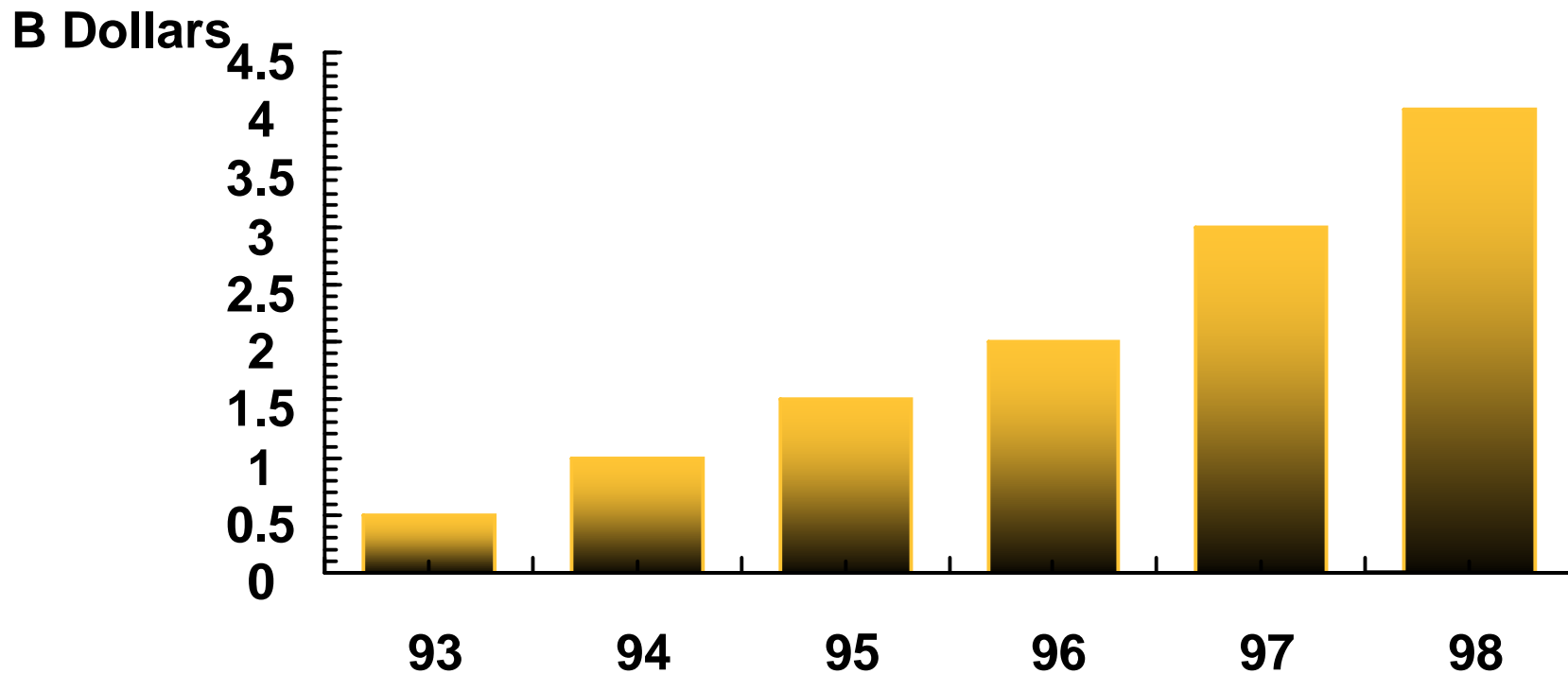
Kevin Wadleigh
kevin_wadleigh@hp.com



Kevin Wadleigh
MSW, TCD
March 31, 2000

Chip production costs per year

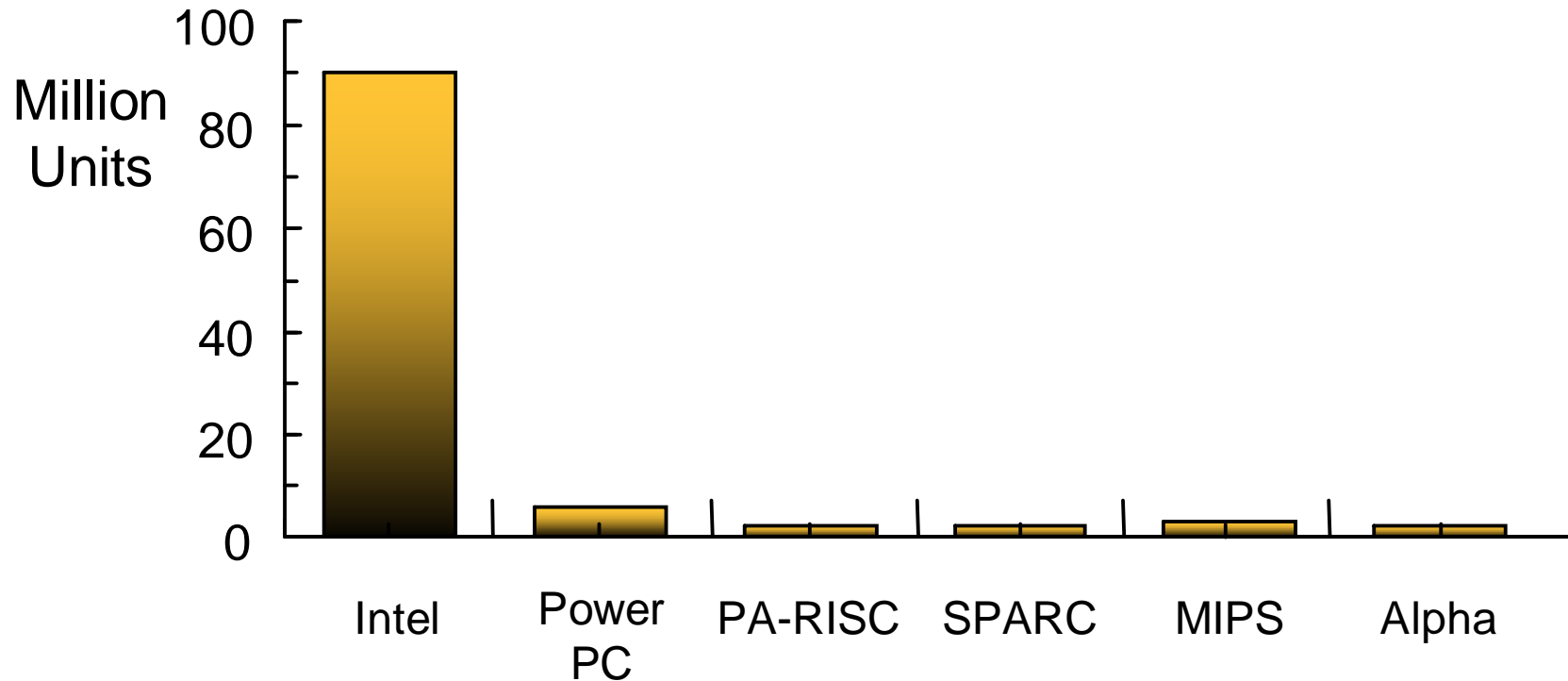
How many proprietary RISC vendors can continue to invest?



(...or, alternatively, face higher chip costs for low volumes with a third party fab?)

Microprocessor Production Capacity

Especially when fabrication and design costs must be recouped against relatively small unit volumes compared with merchants...



Where we've been

Processor
families

CISC
(Complex
Instruction Set
Computing)

Vector

RISC
(Reduced
Instruction Set
Computing)

LIW
(Long Instruction
Word)
Example: EPIC -
Explicitly Parallel
Instruction
Computing

Architecture
(Instruction Set)

IA-32

C-Series

PA-RISC,
MIPS,
Alpha

IA-64

Processors

Pentium III
Xeon

C-4

PA-8500,
R12000,
Alpha21264

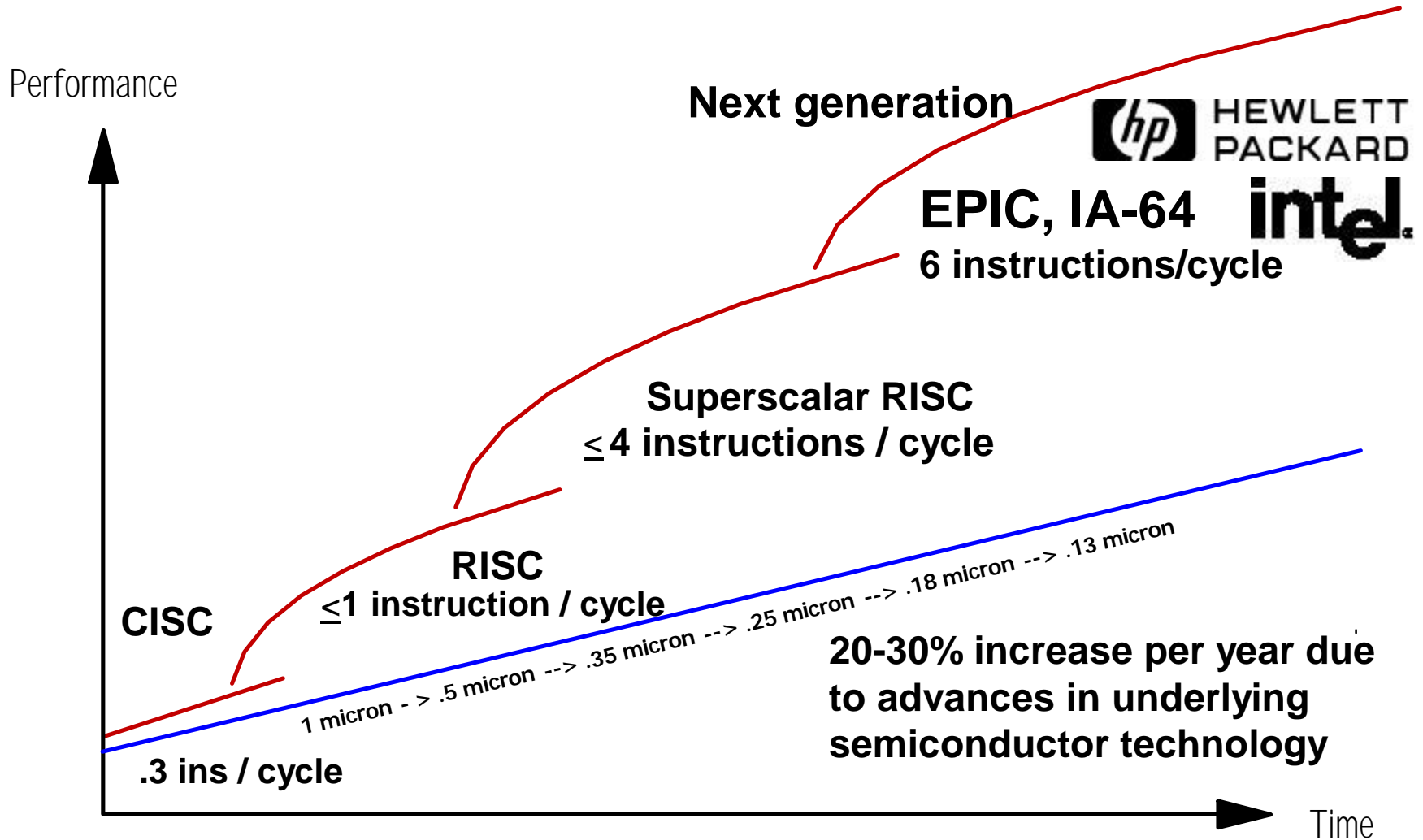
Itanium,
McKinley



Where we're going - IA64

- An *EPIC* story, years in the making
 - HP and Intel jointly designed instruction set
- *Now it can be told*
 - Intel IA-64 home page - <http://developer.intel.com/design/ia-64>
 - Recommended articles:
 - 'Next Generation Instruction Set Architecture' (Crawford, Huck) - <http://developer.intel.com/design/ia-64/next/index.htm>
 - 'Itanium Processor Microarchitecture Overview' (Sharangpani) - http://developer.intel.com/design/ia-64/microarch_ovw
 - The complete (>500 pages) 4 volume 'The IA-64 Architecture Software Developer's Manual' - <http://developer.intel.com/design/ia-64/manuals>

Processor Evolution



EPIC – Why is it better?

- Problems with other processor families
 - Poor instruction parallelism
 - Branches inhibit performance
 - High memory latency
- EPIC attempts to address these by
 - Advanced software (compiler) techniques
 - Designing hardware that allow the compilers to tell the hardware how to improve parallelism, remove branches, decrease latency effects

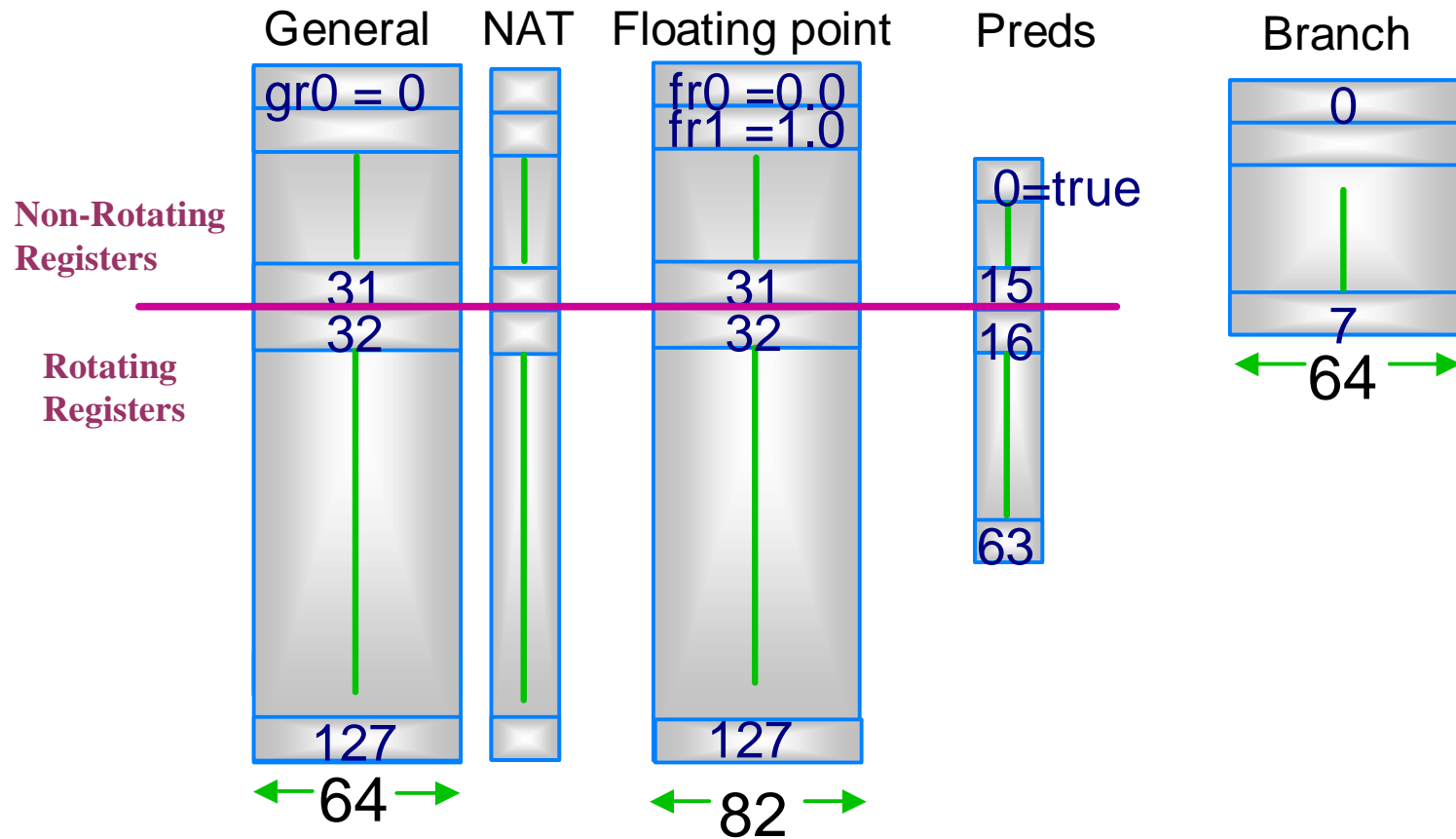
IA-64

IA-64

- Architecture resources
- Predication
- Register rotation
- Speculation
- Processors

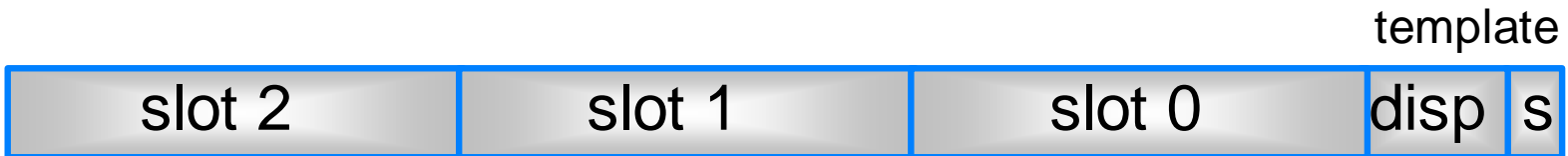


Machine Resources



Instruction Bundling

- 128-bit aligned instruction bundles contain
 - three 41-bit instructions
 - 5-bit template consisting of 4-bit dispersal template + 1 stop bit
- Branches are to bundle boundaries
- Implementations are allowed to have any number of functional units
- Template controls dispersal to functional units: Memory, Integer, Floating-point, Branch, Long immediate



Templates and Dispersal

Templates:

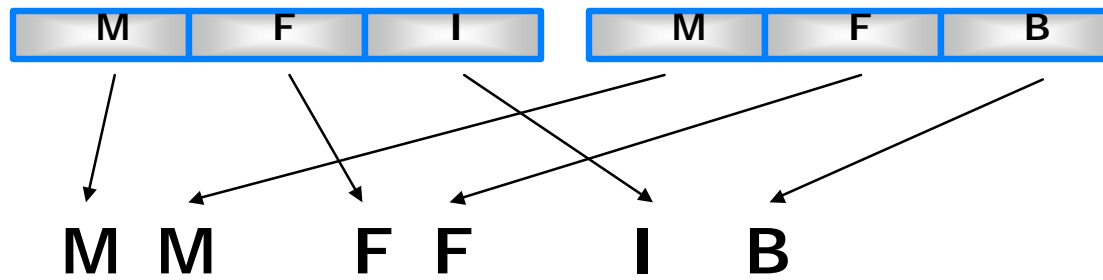
```

0 1 2
M L X
M I I
M I/ I
M M I
M/M I
M M F
M M B
M F I
M F B
M I B
M B B
B B B

```

/ = stop bit

Each template is available
with stop bit at end



Dispersal maps the instructions to functional units. This example shows a CPU that can perform at least two M units, two F units, an I unit and a B unit in one cycle. (Itanium can perform 2M, 2I, 2F, 3B)

Parallelism – Code example $y = x + y$

- Instruction stream

```
ldfd      f10 = [r21]    // load x
ldfd      f11 = [r24]    // load y
;;        // break in parallelism
fadd.d    f10 = f10,f11  // y = x + y
;;        // break in parallelism
stfd      [r24] = f10    // store y
```

- Maps to

```
M        M        nop.l ;; // MMI
nop.m    F        nop.l ;; // MFI
M        nop.l    nop.l  // MII
```

Predication – removes branches

- Converts a control dependence to a data dependence
 - Compare instructions set predicate bit
 - Predicated instructions are either normally executed or they do not affect the architectural state – example code below

```
if (ix .eq. iy) then
```

```
    a = 0
```

```
else
```

```
    c = 0
```

```
endif
```

- Becomes

```
cmp.eq p6,p7 = r16,r17 ;;
```

```
(p6) fadd.d f4 = f0,f0
```

```
(p7) fadd.d f5 = f0,f0
```

- Maps to

```
nop.M      I      nop.I ;;
```

```
nop.M      F      nop.I
```

```
nop.M      F      nop.I
```



Software Pipelining

- Traditional architectures use loop unrolling to hide latencies
 - High overhead: extra code for loop body, prologue, and epilogue
- Synergistic use of IA-64 features allows efficient pipelining
 - Special branches cause registers to rotate
 - Register rotation removes need for explicit unrolling
 - Predicate rotation removes prologue & epilogue

Register Rotation

- Key to good loop performance
 - software pipelining uses register rotation
 - acts like short vectors
 - with each iteration of a loop, data in rotation registers moves to the next register in the set

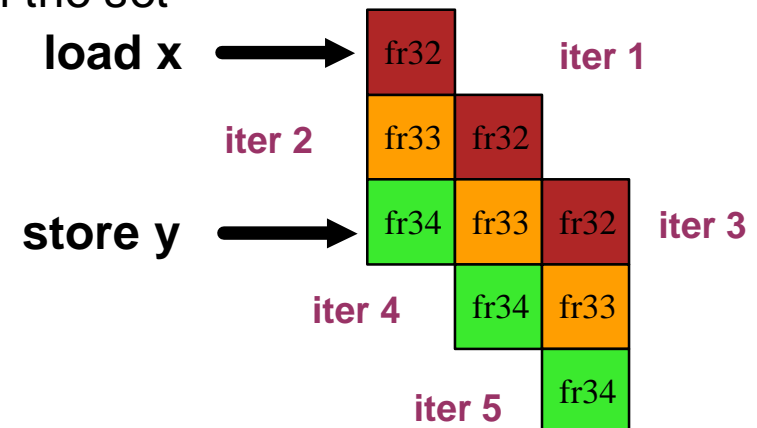
- Code example

```
do i = 1,n
  y(i) = x(i)
enddo
```

- Becomes

```
loop:
  ldld      fr32 = [r26],8 // load x and incr address
  stfd      [r22] = fr34,8 // store y (2 iter after load of x)
  br.ctop.sptk iloop ;; // loop instruction - reg. rotate
```

- This example omits predication necessary for prologue and epilogue

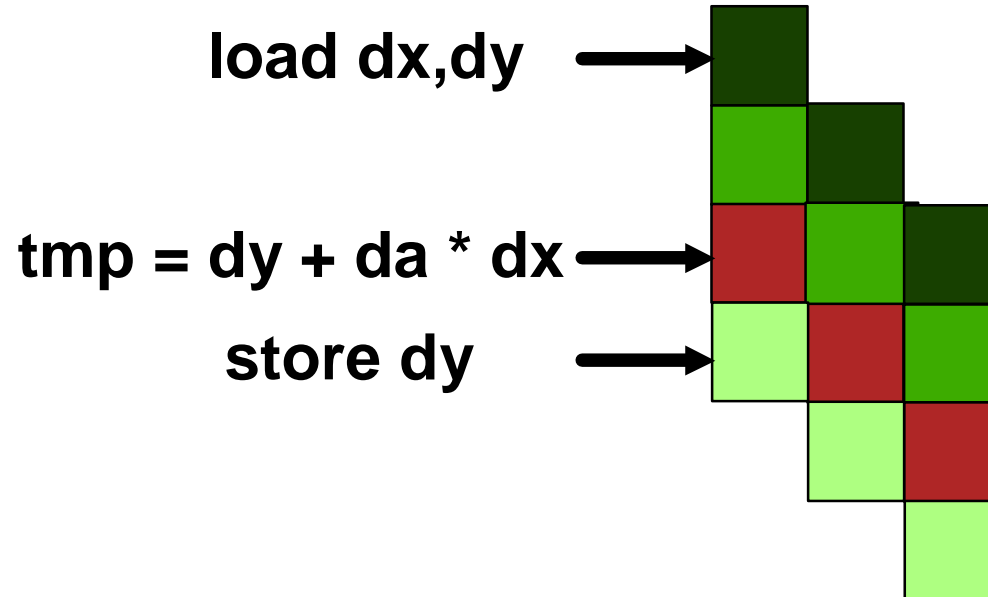


Software Pipelining using Rotation and Predication

- DAXPY inner loop
 - for (i = 0; i < 3; i++)
 - dy[i] = dy[i] + (da * dx[i]);
 - (2 loads, 1 fma, 1 store per iteration)
- Consider a hypothetical processor than can perform
 - 2 loads, 1 fma, 1 store per iteration
 - load latency of 2 cycles
 - fma latency of 1 cycle
- (Itanium can perform: 2M, 2I, 2F, 3B per cycle)

Example: Pipeline

Each column represents 1 source iteration



Example Code

```

.rotf dx[3], dy[3], tmp[2]           // short vectors
    mov    ar.lc = 2                 // lc = loop count
                                        //      = #iterations-1

    mov    ar.ec = 4                 // epilogue count
                                        // #stages (or # pred)

    mov    pr.rot = 0x10000         // p16=1, p17=p18=...=0
    ;;

looptop:
    (p16) ldfd    dx[0] = [dxsp],8
    (p16) ldfd    dy[0] = [dysp],8
    (p18) fma.d   tmp[0] = da, dx[2], dy[2]
    (p19) stfd    [dydp] = tmp[1],8
    br.ctop looptop
    ;;

```

Loop Execution

Execution Sequence

→ (p16) ld_x (p16) ld_y (p18) fma (p19) st

Predicates

16: 1

17: 0

18: 0

19: 0

Loop 1

LC=2 EC=4



Loop Execution

Predicates

16: 1

17: 1

18: 0

19: 0

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st

→ (p16) ld_x (p16) ld_y (p18) fma (p19) st

Loop 2

LC=1 EC=4



Loop Execution

Predicates

16: 1

17: 1

18: 1

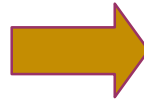
19: 0

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st



Loop 3

LC=0 EC=4



Loop Execution

Predicates

16: 0

17: 1

18: 1

19: 1

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st
 (p16) ld_x (p16) ld_y (p18) fma (p19) st
 (p16) ld_x (p16) ld_y (p18) fma (p19) st
 (p16) ld_x (p16) ld_y (p18) fma (p19) st



Epilogue 1

LC=0 EC=3



Loop Execution

Predicates

16: 0

17: 0

18: 1

19: 1

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

→ (p16) ld_x (p16) ld_y (p18) fma (p19) st

Epilogue 2

LC=0 EC=2



Loop Execution

Predicates

16: 0

17: 0

18: 0

19: 1

Execution Sequence

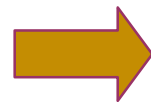
(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st

(p16) ld_x (p16) ld_y (p18) fma (p19) st



(p16) ld_x (p16) ld_y (p18) fma (p19) st

Epilogue 3

LC=0 EC=1



Loop Execution

Predicates

16: 0

17: 0

18: 0

19: 0

Execution Sequence

(p16) ld_x (p16) ld_y (p18) fma (p19) st(p16) ld_x (p16) ld_y (p18) fma (p19) st(p16) ld_x (p16) ld_y (p18) fma (p19) st(p16) ld_x (p16) ld_y (p18) fma (p19) st(p16) ld_x (p16) ld_y (p18) fma (p19) st(p16) ld_x (p16) ld_y (p18) fma (p19) st

 fall through


 Done

LC=0 EC=0



Pipeline and Latency

- Suppose we change to latencies to
 - load latency of 6 cycles
 - fma latency of 4 cycles
- Each column represents 1 source iteration

load dx,dy →

tmp = dy + da * dx →

store dy →



Updated Loop

```

.rotf dx[7], dy[7], tmp[5]

        mov     ar.lc = 2           // #iterations-1
        mov     ar.ec = 11        // #stages
        mov     pr.rot = 0x10000
        ;;

looptop:
(p16) ldfd    dx[0] = [dxsp],8
(p16) ldfd    dy[0] = [dysp],8
(p22) fma.d   tmp[0] = da, dx[6], dy[6]
(p26) stfd    [dydp] = tmp[4],8
        br.ctop looptop
        ;;

```

Rotation: Summary

- Loop pipelining maximizes performance; minimizes overhead
 - Avoids code expansion of unrolling and code explosion of prologue and epilogue
 - Smaller code means fewer cache misses
 - Greater performance improvements in higher latency conditions
- Reduced overhead allows S/W pipelining of small loops with unknown trip counts
 - Typical of integer scalar codes

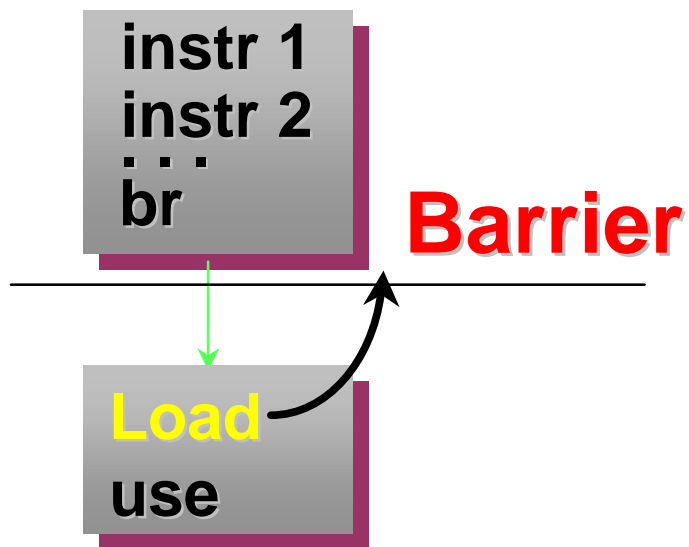
Speculation

- Memory is very far away, so we would like to load data well before its use
- Prefetch instructions will not prefetch pages that have not been mapped by the TLB
- Prefetch instruction will not prefetch data from invalid addresses
- Speculative loads allow users to try to load data from addresses regardless of whether or not the data will be used, the address will be written to in the meantime, or the address is known to be valid. What could go wrong?
- Control speculation versus data speculation
 - Control - moves loads around branches
 - Data - moves loads around stores

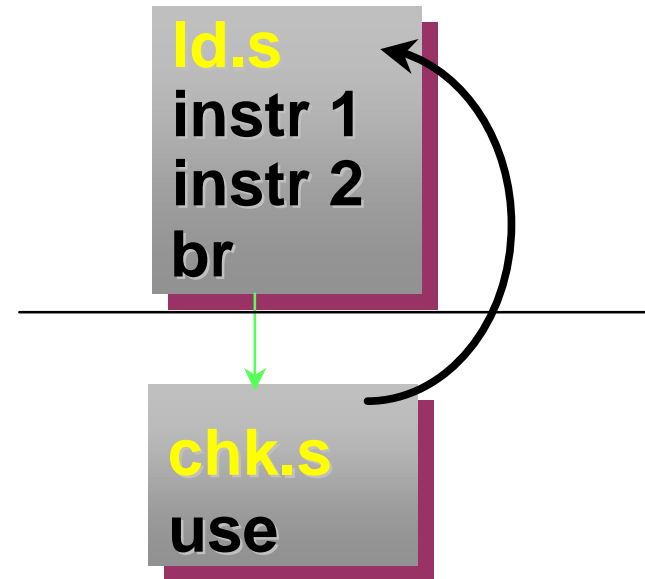
Control Speculation

Move Loads before Branches

Traditional Architectures



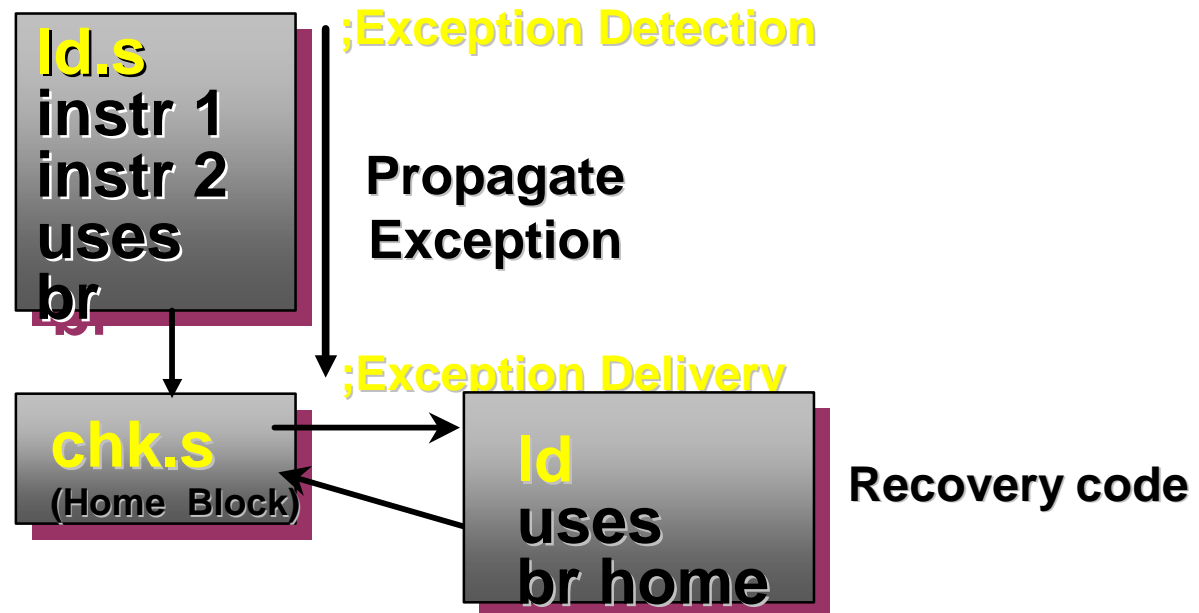
IA-64



Control Speculation

- Regular loads are replaced with speculative load, followed by speculative chk instruction
 - `ld` is replaced by `ld.s, chk.s`
 - `ldf` is replaced by `ldf.s, chk.s`
- For safety, special values are used for illegal returns
 - Integer loads set the Not a Thing (NaT) bit associated with the target general register
 - Floating-point loads set the target floating-point register to a special value: `NaTVal = 0,0x1FFFE,0...0`

NaT (“Not a Thing”) and NaTVal

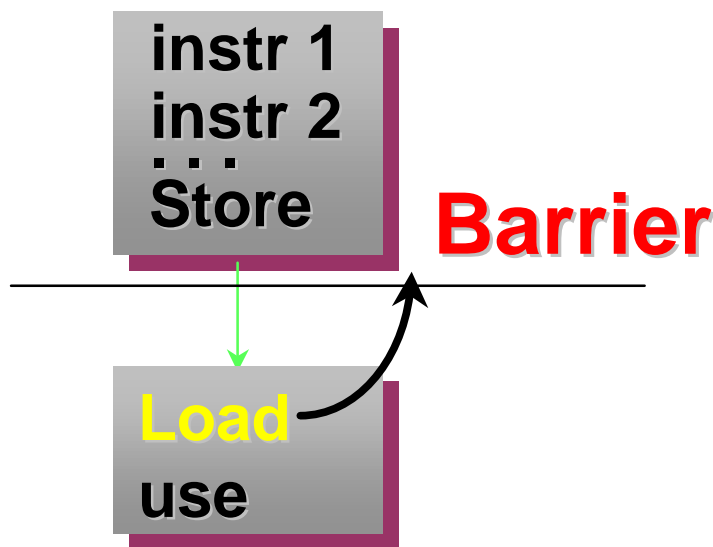


- NaT (or NaTVal) indicates:
 - whether or not an exception has occurred
- If NaT (or NaTVal) set during `ld.s` (`ldf.s`), it is checked by the instruction `chk.s` (usage: `chk.s reg,target`), then branch to target
 - code at target can redo the load and take the normal exception

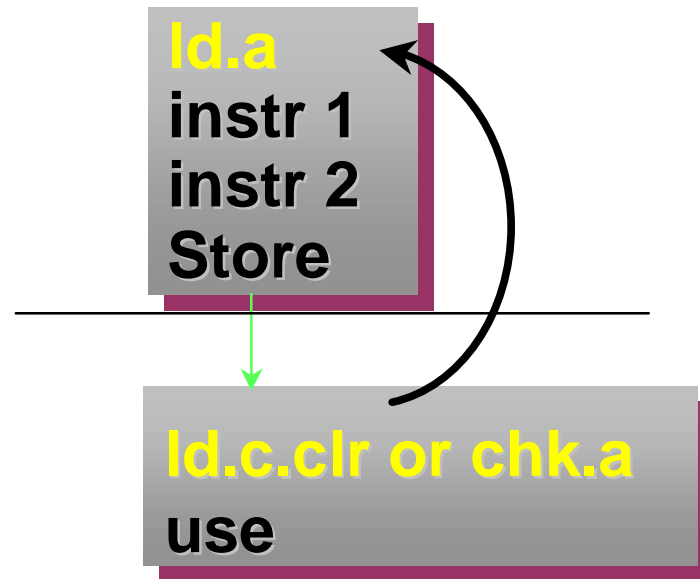
Data Speculation

Move Loads before Stores

Traditional Architectures



IA-64



Data Speculation

- Moves loads around possibly conflicting stores
- Regular loads are replaced with advanced loads, followed by either a check load or advanced chk instruction
- If the only instruction that was ambiguous is the load, then a check load can be performed after the load
 - `ld` is replaced by `ld.a, ld.c.clr`
 - `ldf` is replaced by `ldf.a, ldf.c.clr`
- If there are several instructions that depend on the advanced load, then a `chk.a` can be used to branch to fix up code
 - `ld` is replaced by `ld.a, chk.a`
 - `ldf` is replaced by `ldf.a, chk.a`

Data Speculation - example

- If the only instruction that was ambiguous is the load, then a check load can be performed after the load

```
st      [r4] = r12
```

```
ld      r3 = [r5] ;;
```

- Becomes

```
ld.a   r3 = [r5] ;; // advanced load - note a suffix
```

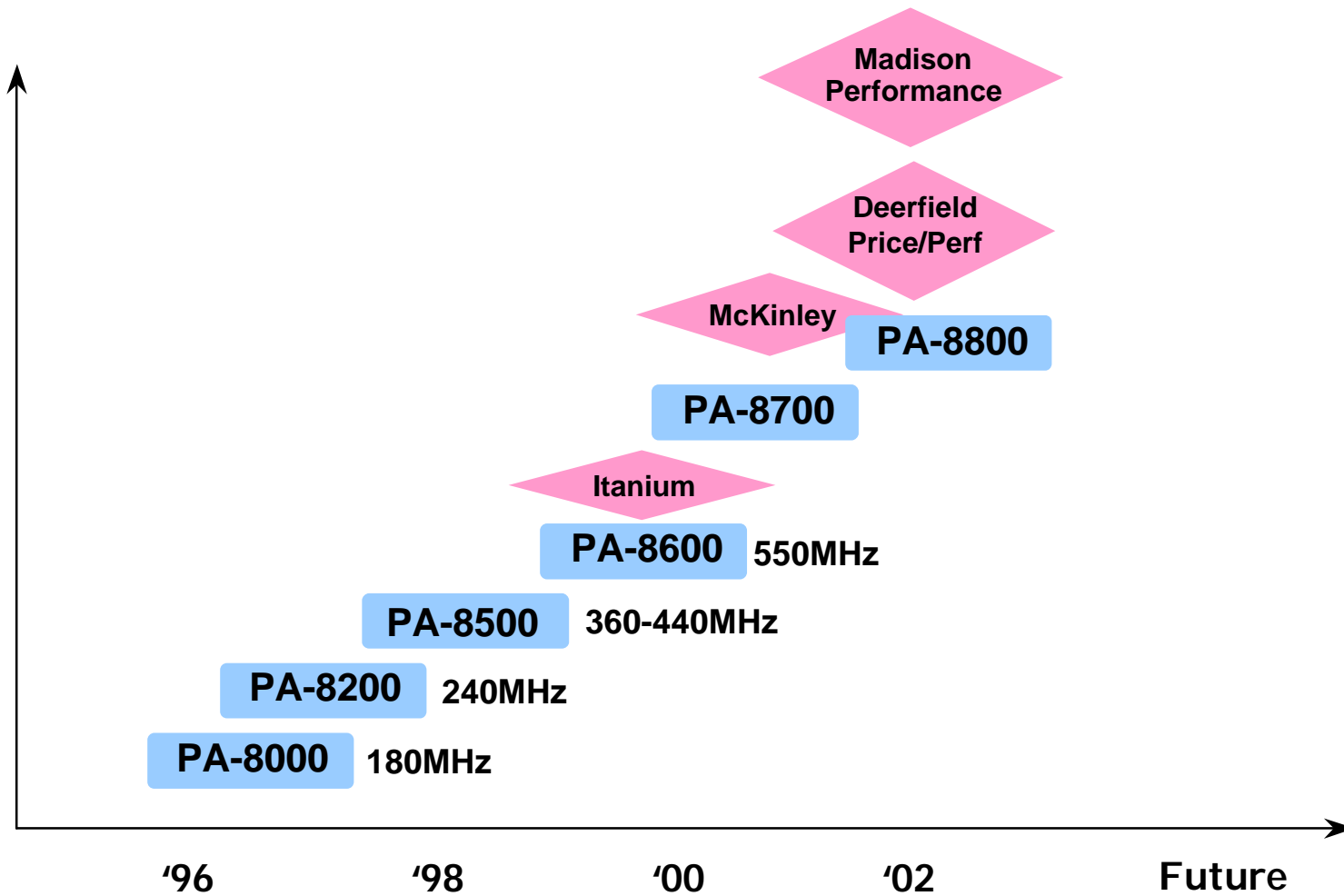
```
...
```

```
st      [r4] = r12
```

```
ld.c.clr r3 = [r5] // if the addr has been modified,  
                // redo it
```

IA-64

HP Microprocessor Roadmap



Kevin Wadleigh
MSW, TCD
March 31, 2000

IA-64 Public Information

- **Itanium**
 - Multiple configurations for servers and w/s
 - Production in mid-2000
 - 0.18m CMOS technology
 - 4 DP Flops/cycle – 3 Gflop/s peak
 - Three level cache hierarchy (64-byte line size)
 - L0: separate instruction and data
 - L1: unified cache on die
 - L2: off die, 2 or 4 MB
- **McKinley**
 - Clock > 1GHz, increased number of execution units, on die L2 cache
 - Increased bus bandwidth
 - Target production: late '01
- **Madison**
 - McKinley follow-on
 - Performance optimized on 0.13m technology
- **Deerfield**
 - McKinley follow-on
 - Price/performance optimized on 0.13m technology

